SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 299–327 (1997)

Research

Program Understanding Behaviour during Enhancement of Large-scale Software

ANNELIESE VON MAYRHAUSER,* A. MARIE VANS AND ADELE E. HOWE
Computer Science Department, Colorado State University, Fort Collins, CO 80523, U.S.A.

SUMMARY

This paper reports on a software understanding field study during the enhancement of large-scale software. The participants were professional software maintenance personnel from industry. The paper reports on the general understanding process, the kinds of actions programmers preferred during the enhancement task, the level of abstraction at which they were working, and role of hypotheses in the enhancement strategies they used. The results of the observations are also interpreted in terms of the information needs of these personnel during the enhancement task. We found that programmers work predominantly at the code and algorithmic levels with differences depending on the stage of the enhancement. They frequently switch between levels of abstraction. The programmers' main concerns are with what software does and how this is accomplished, not why software was built a certain way. These questions guide the work process. There was strong indication that memory (over)load is an issue. This is, of course, related to the size of the software. Information is sought and cross-referenced from a variety of sources from application domain concepts to code-related information, outpacing current maintenance environments' capabilities which are mostly stratified by information sources, making cross-referencing difficult.

J. Softw. Maint., **9**, 299–327 (1997)

No. of Figures: 5. No. of Tables: 9. No. of References: 21.

KEY WORDS: program comprehension; software evolution; abstraction level; information needs; cross-referencing; software maintenance

1. INTRODUCTION

Program understanding has long been recognized as a central activity in a variety of maintenance tasks. To enhance performance, add new features, leverage existing code, etc., a programmer must first understand the code well enough to know what changes are needed, how to make them, and how to integrate new code into existing applications. For larger software products, understanding will, of necessity, be partial.

Most software follows an evolutionary life cycle, meaning that over the lifetime of the

* Correspondence to: Anneliese von Mayrhauser, Department of Computer Science, Colorado State University, 601 S. Howes Lane, Fort Collins, CO 80523, USA. E-mail: avm@cs.colostate.edu

CCC 1040–550X/97/050299–29$17.50

Received 10 February 1997
Revised 23 May 1997

software, enhancements, as well as other modifications, are made. These may include adding a large and significant number of enhancements and new features, or relatively few. Sometimes programmers stay with a product through several enhancement cycles and become familiar with the code. At other times, programmers are asked to enhance code with which they are completely unfamiliar.

In order to learn more about comprehension behaviour when enhancing software, we observed programmers in both categories to see how software professionals, who are experts in the application domain and in the language/platform, went about enhancing a significant software product. The questions we sought to answer were:

(1) What kinds of actions do programmers perform when enhancing code?
(2) Is it possible to identify a specific comprehension process that is common to the subjects and thus indicative of enhancement tasks?
(3) Do programmers use hypotheses as drivers of cognition? What types of hypotheses are used and how are they resolved?
(4) Are there certain types of information programmers tend to look for when enhancing software?
(5) Do programmers follow the Integrated Comprehension mode of von Mayrhauser and Vans (1995a)? Do they switch between its three model components frequently? Is there a preference for a particular model component?

The starting point for our investigation was the premise that, in industry, large-scale programs are the prevalent focus for software comprehension activities. For these, the Integrated Comprehension model (von Mayrhauser and Vans, 1993a, 1993b, 1995a, 1995b) has been shown to model large-scale code comprehension activities as a combination of three comprehension processes at the program (syntax), situation (language independent, algorithmic) and domain (application) levels.

Section 2 of this paper provides background on the Integrated Comprehension model. Section 3 describes the experimental design. It is an observational field study of maintenance programmers, in industry, working on enhancements to software products having at least 40 000 lines of source code. Section 4 reports on the results of the observations with regards to the questions posed above. Section 5 interprets the results in terms of possible improvements for software enhancement activities. Section 6 summarizes our conclusions and provides working hypotheses, based on our results, that should be evaluated with controlled experiments.

2. INTEGRATED COMPREHENSION MODEL

Existing program understanding models agree that comprehension proceeds either top–down, bottom–up, or in some combination of the two. Observations of personnel working with the source code of large-scale software products (von Mayrhauser and Vans, 1993b, 1995a) indicate that comprehension involves both top–down and bottom–up activities. Soloway and Ehrlich's model (Soloway and Ehrlich, 1984; Soloway, Adelson and Ehrlich, 1988) forms the basis for the top–down component (the domain model) while Pennington's

model (Pennington, 1987a, 1987b) inspired the program and situation models. The Integrated Comprehension model consists of the following major components:

(1) program model,
(2) situation model,
(3) top–down model (or domain model), and
(4) knowledge base.

The first three are comprehension processes. The fourth is necessary for successfully building the three models. Program, situation, and top–down (or domain) model building form the three processes that lead to an understanding of code. As Figure 1 illustrates, any of these comprehension processes may be activated from any of the others. Beacons, goals, hypotheses and strategies determine the dynamics of the cognitive tasks and the switches between the models. Each process component contains the internal representation (mental model) of the program being understood. This representation differs in level of abstraction for each model. Each model component also includes strategies to build this internal representation. The knowledge base furnishes the comprehension process with information relevant to the comprehension task. It also stores any new and inferred knowledge.

The *top–down* or *domain* model of program understanding is typically invoked during the comprehension process if the code or type of code is familiar. The top–down model or domain model represents knowledge schemes about the application domain. For example, a domain model of an Operating System (OS) would contain knowledge about the components of an OS (memory management, process management, OS structure, etc.) and how they interact with each other. This knowledge often takes the form of specialized schemes, including design rationalization (e.g., the pros and cons of first-come-first-served versus round-robin scheduling). Obviously, a new OS will be easier to understand with such knowledge than without it. Domain knowledge provides a motherboard into which specific product knowledge can be integrated more easily. It can also lead to effective strategies to guide understanding (e.g., understanding high paging rates requires understanding how process scheduling and paging algorithms are implemented).

Hypotheses are important drivers of cognition in top–down understanding. Letovsky (1986) defines *hypotheses* as conjectures and comprehension activities (actions) that take on the order of seconds or minutes to occur. *Actions* provide a classification of programmer activities, both implicit and explicit, occurring during a specific maintenance task. Examples of action types include ‘asking a question’ and ‘generating a hypothesis’. Letovsky identified three major types of hypotheses:

- *Why* conjectures hypothesize the purpose of some function or design choice.
- *How* conjectures hypothesize about the method for accomplishing a program goal.
- *What* conjectures hypothesize about what something is, for example, a variable or function.

There are degrees of certainty associated with conjectures. They vary from uncertain guesses to almost certain conclusions.

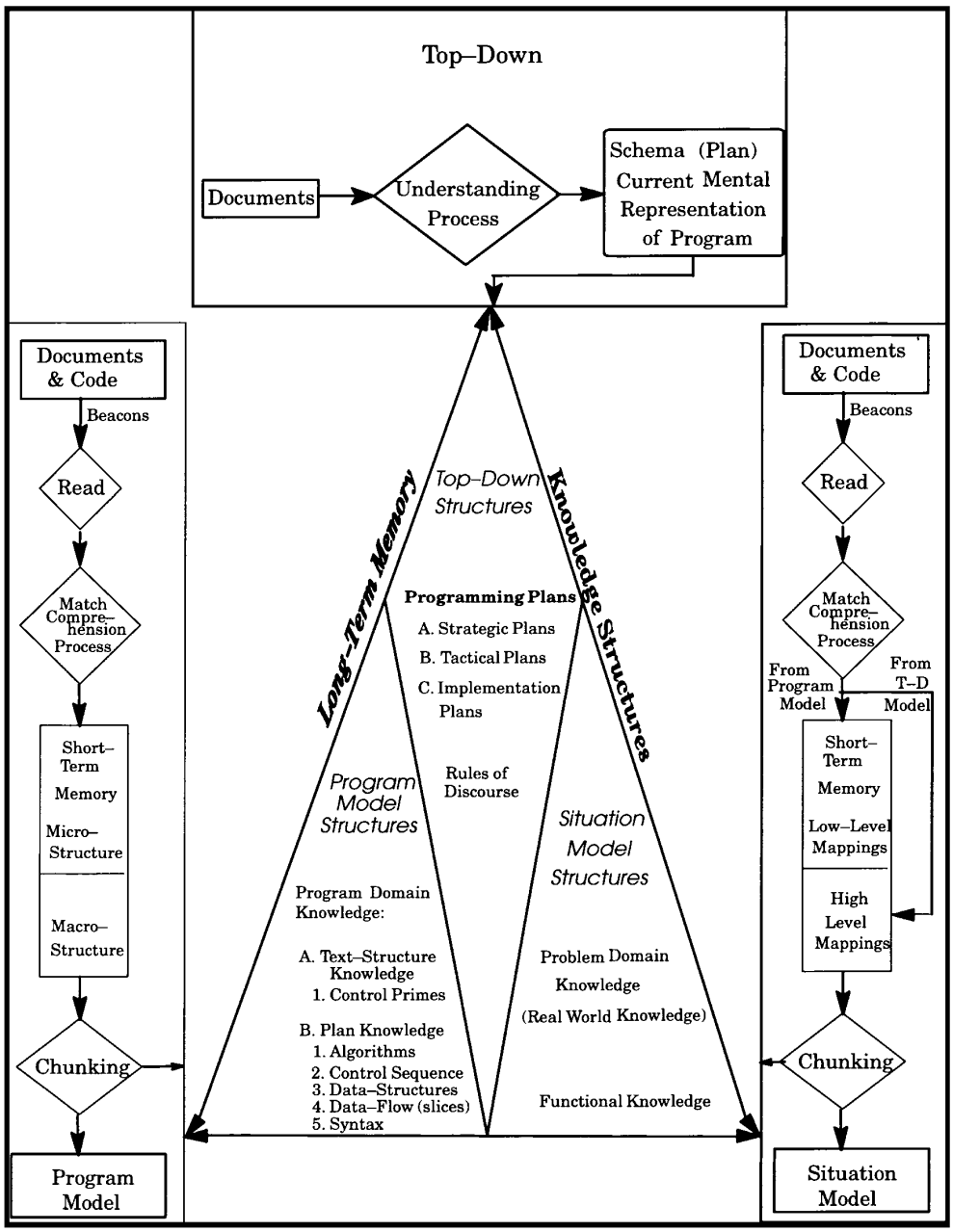

Figure 1. Diagrammatic representation of the Integrated Comprehension model from von Mayrhauser and Vans (1995a)

Brooks (1983) theorizes that hypotheses are the only drivers of cognition. Understanding is complete when the mental model consists entirely of a complete hierarchy of hypotheses in which the lowest level hypotheses are either verified (against actual code or documentation) or fail.

At the top of this hierarchy is the primary hypothesis, a high-level description of the program function. It is necessarily global and non-specific. Once the primary hypothesis exists, subsidiary hypotheses that support the primary hypothesis are generated. According to Brooks, hypotheses cannot be verified until they are close enough to the code or documentation to be compared with program text or documentation. The process continues until the mental model is built. Brooks considers three reasons why hypotheses fail: (1) code to verify a hypothesis cannot be found; (2) confusion due to a single piece of code that satisfies different hypotheses; and (3) code that cannot be explained.

Goals or *questions* embody the cognitive processes by which programmers understand code (Letovsky, 1986). A goal can be explicit as in: '...I have to do a *man* command on the *find* command to determine whether there are any options to *find* that allow me to search for file names...' or inferred from a hypothesis like: 'you really have three libraries involved, I believe, at this point' where the goal is to determine the number of libraries the program uses. Both examples involve the formation of a hypothesis in support of reaching the goal (answering the question). Hypotheses then lead to supporting actions, stating of lower level goals, subsidiary hypotheses, etc. Hypotheses occur at all levels, the application domain, algorithmic and code levels.

When code to be understood is completely new to the programmer, Pennington (1987a, 1987b) found that the first mental representation programmers build is a control flow abstraction of the program called the *program model*. For example, operating system code may be understood by determining the control flow between modules. Then we may select one module for content analysis, e.g., a scheduling module. This may use an implementation of a doubly-linked list. The code representation is part of the program model. The abstraction of the scheduling queue as a doubly-linked list is part of the situation model representation.

Once the program model representation exists, Pennington showed that a *situation model* is developed. This representation, also built from the bottom–up, uses the program model to create a dataflow/functional abstraction. The integrated model also assumes that programmers unfamiliar with the domain first start building a program model. However, to assume that a full program model is built before abstracting to the situation or domain level would create cognitive overload for professionals working on software products with over 40 000 lines of code. Rather, what we expect to happen is the abstraction of program model information at the situation and domain levels as it helps the programmer remember how the program works and what it does.

Programmers switch between any of the three model components during the comprehension process. When constructing the program model, a programmer may recognize clues (called *beacons*) in the code, indicating a common task, such as sorting. For example, a beacon may be the characteristic pattern of value switches indicating a common task (such as sorting), or it may be the name of the function (such as QSORT). If for example, a beacon leads to the hypothesis that a sort is performed, the switch is to the top–down model. The programmer then generates subgoals to support the hypothesis and

searches the code for clues to support these subgoals. If, during the search, a section of unrecognized code is found, the programmer jumps back to building the program model. Figure 1 illustrates the relationships between the three submodels and the related knowledge.

The definition of the Integrated Comprehension model allows a refinement in terms of tasks and task sequences for each of the three comprehension processes. Comprehension is further guided by the systematic bottom–up, the opportunistic top–down, or a mixed systematic/opportunistic strategy.

A systematic approach is one in which the programmer applies an ordered process to understanding code completely, as for example, code comprehension line-by-line. An opportunistic approach involves studying code in an as-needed fashion. The distinction between systematic and opportunistic (or as-needed) is important because Littman *et al.* (1986) found that programmers who use a systematic approach to comprehension are more successful at modifying code (once they understand it) than programmers who take the opportunistic approach. Although the systematic strategy seems better, or safer, it is unrealistic for large programs. A disadvantage of the opportunistic approach is that understanding is incomplete and code modifications based on this understanding may be error prone (Littman *et al.*, 1986). Writing code designed for opportunistic understanding is one solution to this problem. Using tools that help make opportunistic understanding less error prone is another.

Various aspects of the Integrated Comprehension model have been confirmed in prior studies. Thus, von Mayrhauser and Vans (1993b) showed for one enhancement task that the programmer switched between all model components of the Integrated Comprehension model and reported actions occurring at all three levels of the model. Also, von Mayrhauser and Vans (1993a) extended these results to include a debugging task. That study also analysed for detailed action types, rather than merely by component model. Both studies interpret their observations in terms of possible tool capabilities. Later, von Mayrhauser and Vans (1994, 1996b, 1996c) investigated cognition processes to see whether observation could confirm the processes stipulated in the model. Also, von Mayrhauser and Vans (1994, 1996c) reported on the comprehension process of one subject who had used a systematic understanding strategy where actions could be aggregated into episodes, episodes into aggregate processes, and those, in turn, into a session-level process. The analysis found seven types of episode-level processes, three aggregate-level processes, and one session-level process. The episode-level processes include switches between model levels and thus, are not pure bottom–up understanding processes.

By contrast, von Mayrhauser and Vans (1996b) reported on the comprehension process of a programmer who was porting software and employed a fundamentally different comprehension process related to an opportunistic strategy. This process was driven, and could be structured around, a hierarchy of goals, hypotheses and actions. Still later, von Mayrhauser and Vans (1997a, 1997b) reported on comprehension actions of four corrective maintenance programmers. All these results support the Integrated Comprehension model, the switching behaviour between model components, and the role of hypotheses in an opportunistic understanding process.

3. DESIGN OF STUDY

3.1. Programming session

This study reports on an observational field study of professional maintenance programmers working on software enhancements. The software consisted of at least 40 000 lines of code. Each observation involved a *programming session* in which the programmers were asked to think aloud while working on enhancements. We audio and/or video taped this as a thinking aloud report. Sessions were typically two hours long. We observed two professional programmers, both of whom were experts in the language/platform and the application domain of the software. One had acquired significant knowledge about the software product to be enhanced while the other programmer had very little accumulated knowledge about the product. Both were adding new functionality to an existing software product.

3.2. Programmer participants

Programmer EN1 is an expert in the operating systems application domain. He is also an expert with the particular operating system he was assigned to enhance. He has worked with the product for about three and a half years. He is very familiar with the programming language used to implement the operating system.

His task during the programming session was to add new functionality to an existing operating system's command. The enhancement was to be embedded in code not written by EN1. At the start of the session, he had already written the enhancement code and was in the process of removing bugs. Throughout the session he would encounter a bug, track it down, fix it and recompile the code, which usually uncovered another bug. Within the two hours he was able to remove all compile bugs and watch the enhancement work correctly using local tests. At the end of the session he was confident enough in the soundness of the new code to begin regression testing.

Programmer EN2 worked on an enhancement task that involved adding a check for disk-full to a schematic capture CAD system. He was an expert in the domain. The subsystem that handles file creation was the general area where the new code would be added. He was not very familiar with this part of the system. Prior to the programming session he had used a debugger to step through the code so he could get an idea of the structure. The system was written in MainSail, a structured programming language similar to Pascal. EN2 had eight years of experience with MainSail.

The main task EN2 worked on during the programming session was to determine the correct location to add the check for disk-full. The strategy he took was to load the code into the debugger and set breakpoints at strategic locations where he thought the enhancement might be added. He successively set and cleared breakpoints so he could look at details for relevant sections and skip details for those sections he judged unimportant to the task. By the end of the session, he had located where the enhancement should go and had written pseudo-code that would eventually be replaced by real code.

3.3. Tasks

The tasks of both programmers represent real work assignments and are not uncommon in industry. The two tasks represent situations in which programmers find themselves on occasion: being assigned software over a longer term, thus having the opportunity to develop significant knowledge about it, or 'inheriting' a software product in one's area of expertise, but having to develop knowledge about the product itself while enhancing it. Additionally, the subjects worked in different stages of the enhancement task. This provides a more complete picture as each individual observation would have taken longer than two hours.

The tasks observed result in too small a sample size to permit generalizing some of the results. However, since the task specifics of each subject were so different, any commonalities are important to note, since this is a strong indication of a behaviour relevant for many enhancement tasks. In addition, the detailed nature of the observational analysis and the amount of data gathered make it possible to perform statistical analyses and to report the statistical significance of behaviour patterns for each subject. Further, the work reported here adds to prior observations of various types of maintenance tasks summarized in Section 2 (von Mayrhauser and Vans, 1993a, 1993b, 1995a, 1996a, 1996c) in support of the Integrated Comprehension model.

3.4. Protocol analysis

During the tasks, our objectives were to analyse the observations for characteristic activities and behaviour. The analysis also identified information needs. We used protocol analysis for this purpose.

Protocol analysis is used for analysing observational data. Think-aloud reports of subjects working on tasks are transcribed and classified using categories decided on prior to the actual analysis. For example, we expect to find maintenance programmers generating hypotheses and reading code during maintenance. Each statement in the transcript is *encoded* as an instance in one of the a priori categories. Thinking aloud must occur concurrently with the task for the data to be accurate. The analysis proceeds from identifying single actions of various types to determining action sequences and extracting cognition processes and strategies. The analysis parallels experimental work related to the Integrated Comprehension model described in Section 2 (von Mayrhauser and Vans, 1993a, 1993b, 1995a). That model supplies a framework for discovering the role of actions, hypotheses and comprehension behaviour in large-scale code understanding. Protocol analysis proceeded in the same steps (see Figure 2) used in prior studies (von Mayrhauser and Vans, 1993a, 1993b, 1995a, 1996a, 1996c) and described below.

Step 1: actions

The first analysis on the protocols involved *enumeration of action types* as they relate to the Integrated Comprehension model. Action types classify programmer activities, both implicit and explicit, during a specific maintenance task. Examples of action types are: 'generating hypotheses about program behaviour' or 'mental simulation of program state-

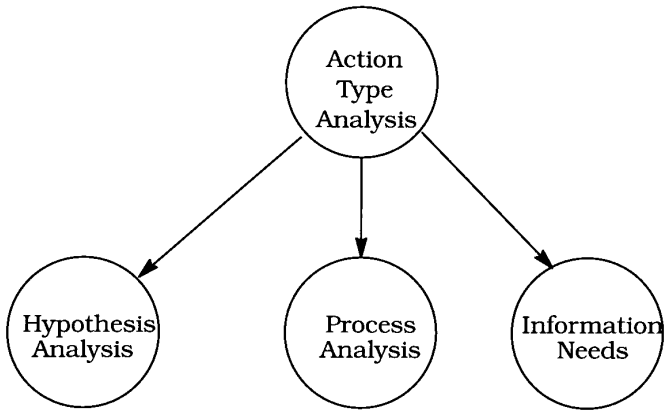

Figure 2. Overview of protocol analysis and research process

ment execution'. The list of actions used (cf. Table 3) was the same as that developed by von Mayrhauser and Vans (1993b, 1996c). The development of this list of action types started with a list of actions from Vessey (1985) that were added to based on action types encountered in a protocol used for a pilot study.

The results of this analysis are traces of action types as they occurred in the protocol. Summary data were then computed as frequencies for individual action types as well as cumulative frequencies of actions per model component. The summary data were analysed using chi-square tests on contingency tables to determine whether there were statistically significant differences between the two subjects EN1 and EN2 and prior results for corrective maintenance (von Mayrhauser and Vans, 1997a) and reverse engineering tasks (von Mayrhauser and Vans, 1996a) with regards to action type frequencies.

Step 2: segmentation and information needs

The next step in the analysis combines *segmentation* of the protocols and identification of information and knowledge items. Segmentation classifies action types into those involving the domain (top–down), situation or program model that can be thought of in terms of different levels of abstraction in the mental model. *Information needs* are information and knowledge items that support successful completion of maintenance tasks.

Protocol analysis is an iterative process. A first-pass analysis results in a high-level classification of programmer actions as either program, situation or top–down model components of the Integrated Comprehension model. This is necessary because similar actions appear in different component processes. For example, hypotheses may be generated while constructing any of the three components. Once actions are associated with a particular model component, the next pass identifies the action types of the specific maintenance tasks. Once the action types are identified, the transcripts are re-analysed and encoded using these types as tags on the programmer utterances. An utterance is a verbalization by the programmer during a programming session and captured in the transcripts. Information needs are determined by associating specific clusters of actions with the information need related to it. Table 9 shows action type clusters associated with

each information need. This analysis process differs from the one used earlier by von Mayrhauser and Vans (1993a, 1993b, 1995a) which was much less systematic.

Table 1 contains example protocols to show action type classification and information needs identification. The upper part of Table 1 applies only to action type classification. The two actions are at the program model level. One generates an hypothesis about variable meaning, the other represents a comprehension action (supporting a hypothesis). The lower part of Table 1 deals with identifying information needs. Counting frequencies of information needs indicates the relative importance of the needs for the enhancement tasks.

Step 3: hypotheses

In this analysis, we were particularly interested in actions relating to stating goals, making and resolving hypotheses, and actions supporting goal or hypothesis resolution. We distinguished between three action types (stating a goal, stating an hypothesis, and doing supporting actions) at the three levels of the Integrated Comprehension model (program, situation and top–down or domain levels).

Hypotheses are associated with model components. They are initially identified during action type analysis. Action type analysis tags generation, confirmation or failure of each hypothesis. Each hypothesis is then classified by how it was stated in the protocol. For example, the hypothesis 'warnings are related, we believe, to the resources either not being installed properly or not being compatible with this system' is classified as an

Table 1. Examples of protocol analysis action types and information needs

Analysis type	Tag	Action type	Example protocol
Action-type classification	SYS8	Generate hypothesis (program model)	'...and my assumption is that "nil" with a little n and "NIL" with a big N are equivalent at the moment...'
	SYS7	Chunk and store knowledge (program model)	'...so clearly what this does is just flip a logical flag...'

Analysis type	Tag	Information need classification	Example protocol
Identifying information needs	I9	List of browsed locations (OP2, OP7, SITKNOW, OPKNOW, SYS10)	'...because it calls SPOOL-INTO, yeah. So I was at the right place. A long time ago...'
	I43	General classification of related functions (OP3, OP4, OP14, OP15, OP20, SIT4, SYS8, SYS11)	'Yeah, this is the one. DbFile insert. Insert dbfile. Get the dbfile in. Curse the author of this one. Try to insert defile. So...it sounds a lot like the other one...'

hypothesis about the cause of the buggy behaviour. We classified the *type* of each hypothesis according to Letovsky's taxonomy into *what*, *why* or *how* hypotheses (Letovsky, 1996).

A separate analysis follows each hypothesis through the protocol until it is resolved through confirmation, failure or abandonment. An hypothesis is verbally confirmed or rejected (i.e., it fails). Hypothesis abandonment can either be explicit or implicit. Explicit abandonment occurs when the programmer decides it is not relevant or will be too much trouble to verify. Implicit abandonment occurs when the hypothesis is stated but the programmer never returns to it. We can only conjecture that the hypothesis was either forgotten or dismissed without verbal confirmation.

We counted frequencies of hypotheses at each model level, constructed contingency tables and analysed the contingency tables using a chi-square test to determine whether the two subjects show statistically significant differences in making hypotheses. These frequencies were also compared with the results for porting software (von Mayrhauser and Vans, 1996b) and to the performance of corrective maintenance subjects (von Mayrhauser and Vans, 1997b).

Step 4: process analysis

For actions, process analysis determined the nature of actions over time, graphically. Each action has already been classified by level of abstraction (program, situation or domain level). They are now plotted as a function of 'time' (in terms of numbers of actions). This graph illustrates both how long (in terms of actions) a programmer spends in each model, as well as the frequency of switches and whether switching is fairly unidirectional (top–down or bottom–up) or not.

Switching frequencies and the action trace are also examined statistically. First, a dependency analysis (Howe and Cohen, 1994) between pairs of actions identifies statistically significant patterns of switching behaviour for the subjects. Second, a chi-square test determines whether the switching behaviour differs significantly between EN1 and EN2, as well as between them and the corrective maintenance (von Mayrhauser and Vans, 1997a) and reverse engineering subjects (von Mayrhauser and Vans, 1996a).

This analysis only examines dependencies between pairs of actions. A more comprehensive analysis is to perform State Transition Dependency Detection (STDD) (Howe and Somlo, 1977). STDD automatically constructs state transition diagrams for discrete event sequences. It works by consistently combining statistically significant short patterns into a larger, unified model that illustrates the interrelationships.

4. RESULTS

4.1. Programmer actions

Table 2 shows how often the subjects perform actions at the three levels of abstraction defined in the Integrated Comprehension model. The percentages show the relative frequency of these actions for each subject. Table 2 consists of two contingency tables,

Table 2. Total and relative frequencies of action types by model

Observation and analysis	Top–down model	Program and situation models			Total of actions
		Combined	Situation	Program	
EN1 observations	35	282	62	22	317
enhancement	11%	89%	20%	69%	100%
EN2 observations	154	288	54	234	442
enhancement	35%	65%	12%	53%	100%
Total observations	189	570	116	454	759
enhancement	25%	75%	15%	60%	100%
Chi-square results	$\chi^2 = 55{\cdot}9$ $p < 0{\cdot}0001$		$\chi^2 = 56{\cdot}9$ $p < 0{\cdot}0001$		

with a subject code column on the far left, and a total column on the far right. The first contingency table is 2×2 in size. It has columns of counts for the top–down model and the combined counts for the program/situation models. The second contingency table is 3×2 in size. It picks up the first column of counts from the first table, and breaks apart into two columns the program and situation model counts (that are combined in the first contingency table).

The Table 2 column for the combined program and situation models roughly corresponds to Pennington's (1987a) comprehension model. We wanted to identify patterns based on differences between Pennington's bottom–up model and the top–down model. Programmer EN1 (who knew more about the software) worked mostly at the program and situation model level (89% of the actions). This reflects his ability to connect quickly to the code level, but also reflects that his activities included debugging the enhancement. In this way, his behaviour was similar to the analysis results for corrective maintenance (von Mayrhauser and Vans, 1997a). EN1 also spent a great deal of time in the program model because he was interested in very specific parts of the code he had recently enhanced. Since he was a domain and language expert and had significant experience with the code, we surmise that his task did not require building a top–down mental model. He had worked with this code for several years and, we assume, already had a working top–down mental representation of the code. The task he was working on did not require much adjustment to his top–down mental representation.

The second programmer (EN2) reflects an earlier stage of enhancement work which requires understanding the nature of the enhancement as well as finding the proper location in the code for the enhancement. Not surprisingly, the number of actions at the domain level is higher than for EN1. Even so, both subjects showed more actions at the situation and program model levels than at the domain level.

The two programmers should therefore show different behaviour in terms of the level of abstraction (the component of the Integrated Comprehension model) at which they are

working. We found a significant difference between the subjects ($p < 0.0001$) in chi-square analyses of action-type frequencies. Table 2 shows this analysis for top–down versus combined program/situation model level ($\chi^2 = 55{\cdot}9$; $p < 0.0001$), and for each of these three model components separately ($\chi^2 = 56{\cdot}9$; $p < 0.0001$).

Let us consider the types of actions at each model level next, to see whether this provides more insight. The types and frequency of programmer actions reflect how often programmers worked at each level of abstraction during their attempt to understand the code well enough to be able to make enhancements to code (see Table 3).

Consider totals for both subjects first (rightmost column). The most commonly found action during top–down model construction is use of top–down knowledge (OPKNOW). Top–down knowledge is previously acquired knowledge at the domain level that is elicited from long-term memory. Frequent use of top–down knowledge supports the hypothesis that programmers who are familiar with the application domain develop and use the top–down model to structure information about the system (like a 'motherboard' into which they plug new knowledge or which acts as a structuring device for using existing knowledge). It is much easier to decompose the program into functional units if the programmer is well acquainted with the domain. Hypothesis generation (the third most common action found in the protocols) drives the decomposition process.

The second most frequent action is to generate a task. This fits well with the comprehension process found for porting tasks by von Mayrhauser and Vans (1996b). This process is based on hierarchical sequences of Goal–Hypothesis–Action triads. A comprehension process starts with a goal (what is to be understood or done), supporting hypotheses are made which then lead to one or more of the following:

(1) formulation of subgoals,
(2) formulation of sub-hypotheses, and
(3) actions (tasks) that help to confirm or refute hypotheses.

This interpretation appears reasonable, if we consider the two observations taken together as a 'complete' enhancement task and each individual observation as one part of it. Conversely, behaviour for designing and locating where to make the enhancement versus making and debugging it, shows different action type preferences. EN1 uses fewer action types (using domain knowledge—OPKNOW—and determining sequence—OP2—are the most frequent). EN2, on the other hand, uses more and a larger variety of actions. This reflects his need to build knowledge about the software's functionality and where and how to make the enhancement.

Program-model building typically involves generating hypothesis, examining code and chunking/storing knowledge. (A *chunk* is a knowledge structure consisting of various levels of abstractions of logical blocks of code.) Looking at cumulative counts of action types for both subjects at the program model level, chunking and storing knowledge (SYS7) and referring to previously acquired knowledge (SYSKNOW) are the two most frequent action types. Generating hypotheses is third (SYS8).

Considering each programmer separately, the top three action types for EN1 are the same as for the cumulative data. For EN2, the top three are chunking/storing knowledge (SYS7), considering different code changes (SYS15), and generating new tasks (SYS11).

Table 3. Action types observed for the enhancement process

Code	Action type	EN1	EN2	Total
OP1	Gain high-level program overview	1	1	2
OP2	Determine next program segment to examine	9	6	15
OP3	Generate/revise hypothesis re: functionality	2	6	8
OP4	Determine relevance of program segment	0	8	8
OP6	Determine understand strategy	0	5	5
OP7	Investigate oversight	2		2
OP8	Failed hypothesis	1	1	2
OP9	Mental simulation	0	1	1
OP11	High-level change plan/alternatives	0	17	17
OP13	Study/initiate program execution	0	10	10
OP14	Compare program segments	0	3	3
OP15	Generate questions	0	7	7
OP16	Answer questions	0	4	4
OP17	Chunk and store knowledge	0	13	13
OP20	Generate task	0	35	35
OPCONF	Confirm hypothesis	1	6	7
OPKNOW	Use of top–down knowledge	19	31	50
Total	Top–down model actions	35	154	189
SIT1	Gain situation knowledge	3	1	4
SIT2	Develop questions	2	1	3
SIT3	Determine answers to questions	1		1
SIT4	Chunk and store	7	31	38
SIT5	Determine relevance of situation knowledge	1		1
SIT6	Determine next information to gain	0	1	1
SIT7	Generate hypothesis	4	1	5
SIT8	Determine understand strategy	1		1
SIT9	Determine if error exists (missing function)	2		2
SIT10	Failed hypothesis	1		1
SIT11	Mental simulation	2	11	13
SITCONF	Confirm hypothesis	3	1	4
SITKNOW	Use of situation model knowledge	35	7	42
Total	Situation model actions	62	54	116
SYS1	Read introduction code comments/related documents	2	13	15
SYS2	Determine next program segment to examine	6	10	16
SYS3	Examine next module in sequence	14	11	25
SYS4	Examine next module in control flow	11	17	28
SYS5	Examine data structures and definitions	2	1	3
SYS7	Chunk and store knowledge	17	40	57
SYS8	Generate hypothesis	30	11	41
SYS10	Determine understand strategy	9	3	12
SYS11	Generate new task	14	24	38
SYS12	Generate question	8	7	15
SYS13	Determine if looking at right code	2	3	5
SYS14	Change direction	2		2

Table 3. Continued

Code	Action type	EN1	EN2	Total
SYS15	Generate/consider different code changes	3	25	28
SYS16	Answer question	1	1	2
SYS17	Add/alter code	15	10	25
SYS18	Determine location to set breakpoint	11	6	17
SYS19	Failed hypothesis	10	2	12
SYS20	Determine error/omitted code to add	7	17	24
SYS21	Mental simulation	7	13	20
SYSCONF	Confirm hypothesis	17	9	26
SYSKNOW	Use of program model knowledge	32	10	42
Total	Program down model actions	220	234	454

This reflects well the type of work EN2 was doing—namely, trying to determine how and where to make the enhancement. In the process he must have acquired significant knowledge at the program level (he had limited knowledge of it at the start) and chunked and stored it away. This would explain the high number of SYS7 actions.

At the situation-model level, EN1's preferred action is to use previously acquired knowledge (reflecting his high level of familiarity with the software). Again conversely, EN2's is to chunk and store knowledge at that level, pointing to his acquisition of such knowledge, which is necessary so he understands how to make the enhancement.

4.2. Processes

Switches occur between all three models (top–down, program and situation models). Table 4 summarizes switches between models during the enhancement task. The rows

Table 4. Total and relative frequencies of action switches

Model switched from	Model Switched To			Total switched from
	Top–down	Situation	Program	
Top–down	N/A	15 5%	77 26%	92 31%
Situation	28 10%	N/A	51 17%	79 26%
Program	64 21%	64 21%	N/A	128 43%
Total switched to	92 31%	79 26%	128 43%	299 100%

represent starting models and the columns represent ending models. Typically, switching between program and situation models happens because the programmer is trying to link a chunk of program code to an algorithmic description in the situation model (a switch from program to situation model). Alternatively, the programmer may be looking for a specific set of program statements to verify the existence of some functionality (a switch from situation to program model).

This switching behaviour confirms the assumptions made in the Integrated Comprehension model (switching occurs at all three levels, implying that understanding is neither top–down nor bottom–up, but always a combination). The specifics of the switching behaviour are different from those for the re-engineering subjects (von Mayrhauser and Vans, 1996a) who showed more of a preference for bottom–up switches (from program to situation model). It is also different from corrective maintenance (von Mayrhauser and Vans, 1997a) where the pattern shows fairly even switching rates. For the enhancement tasks, the subjects switched between situation and top–down models significantly less than between the other model component. If the situation model acts as a bridge between the other two, software enhancement tasks may not need it as much. The subjects preferred to make direct connections between the top–down and the program model.

A chi-square test on these data and on frequencies from previous studies shows that the switching behaviour of the subjects working on enhancement tasks is statistically different from the behaviour found by von Mayrhauser and Vans (1997a) for the group doing corrective maintenance ($\chi^2 = 93{\cdot}03$, $p < 0.0001$), and from the behaviour found by von Mayrhauser and Vans (1996a) for the group performing a reverse engineering task ($\chi^2 = 13{\cdot}76$, $p < 0.001$).

Table 5 shows, for both programmers, which action pairs (switches or transitions) are statistically significant (not likely to be due to chance). The first two columns list the action pairs by component model of the Integrated Comprehension model. The third and fourth columns report the significance level of the pairwise pattern for EN1 and EN2

Table 5. Dependency analysis results

Model action pair		Probabilities for programmer	
Model from	Model to	EN1	EN2
Top–down	Top–down	0·00000085	0·00000062
	Situation	—	0·00038
	Program	0·049	0·0000053
Situation	Top–down	0·12	—
	Situation	0·00000036	0·09
	Program	0·00000057	0·11
Program	Top–down	0·00062	0·000006
	Situation	0·00000085	0·000013
	Program	0·00000074	0·00085

respectively. A blank cell indicates that this pattern was not statistically significant and is likely due to chance. There is one blank cell for each programmer. For EN1, it is the transition from domain to situation model; for EN2, it is the transition from situation to domain model. Additionally, the transitions of situation to top–down for EN1, and situation to situation and situation to program for EN2 are of questionable significance due to their relatively high p values of 0·12, 0·09 and 0·11, respectively.

Next, we used State Transition Dependency Detection (Howe and Somlo, 1997) to construct a model of the transitions or switches between the domain, situation and program models. Figure 3 shows the results of this analysis. State Transition Dependency Detection constructs a state transition diagram from statistically significant patterns found in the data. A threshold α, indicates the maximum probability that a particular pattern might have been observed due to noise or chance; thus, an estimated probability below α means

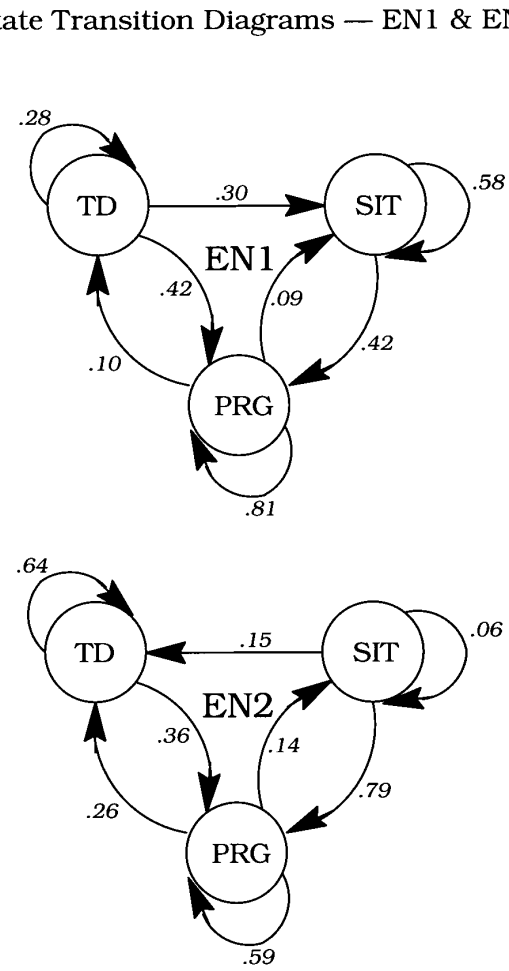

Figure 3. State transition diagrams

that the pattern was *not* likely to have appeared due to noise. Rare patterns are likely to be significant; so, the diagram includes even low transition probabilities. The analysis was done with a threshold value of $\alpha = 0{\cdot}15$ (so p must be $< 0{\cdot}15$ for the pattern to be included) for dependencies in the action trace of length two. The diagrams for the two programmers are different. Most strikingly, EN1 shows switches from the top–down to the situation model, but not the other way. EN2's diagram reveals just the opposite behaviour. EN1 also has far fewer transitions from the program to the situation model indicating less bottom–up understanding. Following up on this analysis, analysing contingency tables comparing transitions for each state in the diagram shows significant differences between the two subjects (TD: $\chi^2 = 14{\cdot}25$, $p < 0{\cdot}0008$; SIT: $\chi^2 = 28{\cdot}4$, $p < 0{\cdot}001$; PRG: $\chi^2 = 25{\cdot}4$, $p < 0{\cdot}001$).

Consider the sequences of actions next. Figures 4 and 5 illustrate how these switches happen over time. The graphs show a line from one model component to another when the programmer switches from an action in one model to another. Thus, the inclination of the line represents how long a programmer stayed at a level before switching: a very steep line indicates few actions between switches, longer numbers of actions between switches are indicated by a shallow incline or decline in the switch line.

The two programmers show very different patterns in these models of behaviour. EN1

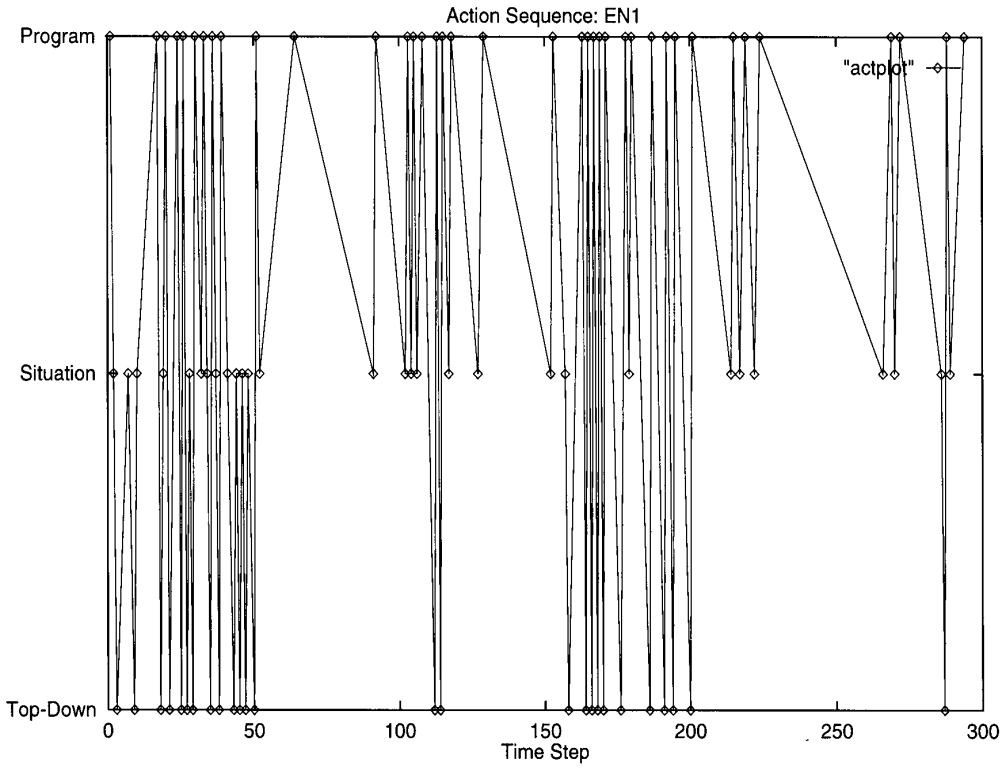

Figure 4. EN1 action sequence

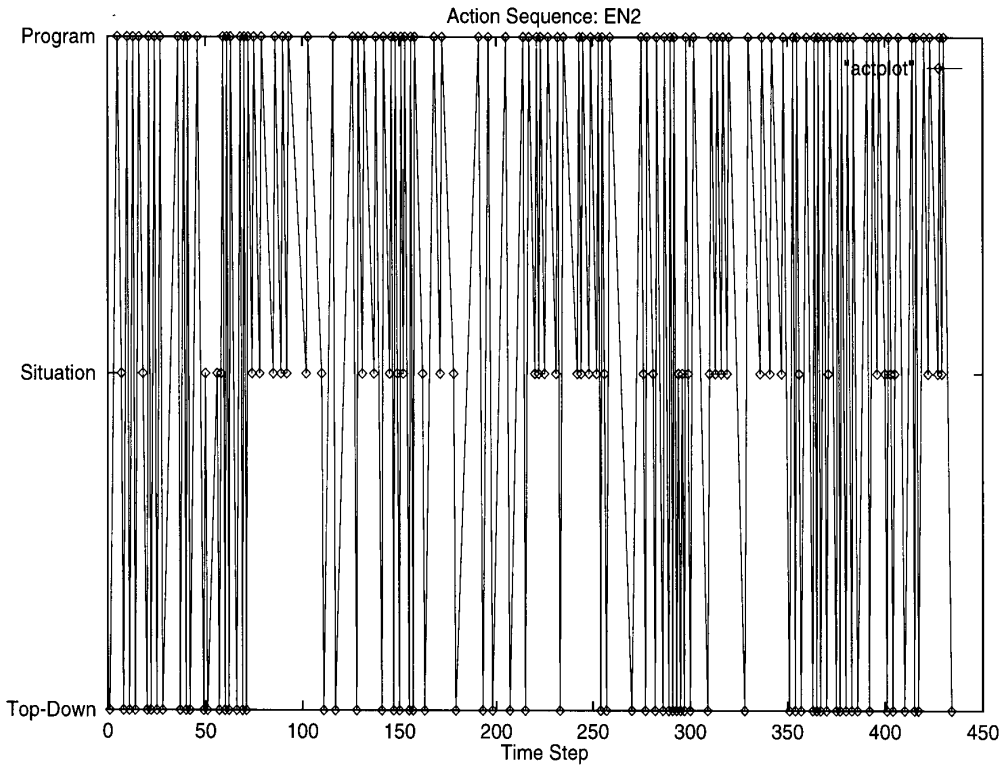

Figure 5. EN2 action sequence

alternates periods of short numbers of actions at the program or top–down model, switching between the two, with a larger number of actions at the program level and occasional connections to the situation model level.

EN2, on the other hand, shows much shorter action sequences at any one level and a great deal more cross-referencing. This may be due to his being in an earlier stage of the enhancement task and/or not having worked with the software product as long and thus, having less knowledge about it.

4.3. Hypotheses

Table 6 lists all hypotheses observed at each level of abstraction in the Integrated Comprehension model (von Mayrhauser and Vans, 1995a). The leftmost column identifies at which model level the hypothesis is made, and the second column gives the tag used to code the hypothesis type. The third column describes the theme of the hypothesis type. The next three columns identify the frequency of each hypothesis type, first cumulatively, then for EN1 and EN2 individually. The hypothesis types in Table 6 represent a refinement of the hypothesis classes used in von Mayrhauser and Vans' (1996b) paper.

Table 7 reports, for each programmer, the number of hypotheses made at each model

Table 6. Frequencies observed for hypothesis types

Model	Tag	Hypothesis type	Total references	EN1	EN2
Top–down (domain)	OPH1	Domain procedure functionality/concepts	9		9
	OPH5	Existence of installed (running) program	2		2
	OPH9	Permissions/environment set correctly/tool functionality	1	1	
Model	OPH10	Location to add functionality	3		3
	OPH11	Comparison of functionality at high level	2		2
	OPH16	Level and structure of code/scope	1	1	
	Total	Top–down model hypotheses	18	2	16
Situation Model	SITH2	Function/code block execution order/state	1	1	
	SITH3	Function/procedure function, call function	2	1	1
	SITH4	Effect of running program	1	1	
	SITH7	Existence of functionality/algorithm/variable	1	1	
Model	Total	Situation model hypotheses	5	4	1
Program model	SYSH2	Function/procedure function	4	1	3
	SYSH5	Location/type/existence of function call	3	2	1
	SYSH6	Statement execution order/state	3	3	
	SYSH7	Variable value/defaults	8	7	1
	SYSH8	(Non-)existence of construct (var/code)	1	1	
	SYSH13	Code block/procedure comparison	1		1
	SYSH16	Code correctness, cause/location of error	16	15	1
	SYSH18	Location to add code/alternatives	5	1	4
	Total	Program model hypotheses	41	30	11

level. The organization of Table 7 follows the pattern used in Table 2. A chi-square test on the data in Table 7 shows significant differences ($p < 0{\cdot}0001$) between the two subjects in how many hypotheses were made at each level. First, enhancement appears to need fewer hypotheses than other tasks. EN1 and EN2 show 28 and 36 hypotheses, while corrective maintenance subjects (von Mayrhauser and Vans, 1997b) made an average of 51 hypotheses each. EN1 made most of the hypotheses at the program model level. He made almost no hypotheses at the situation and domain model level. This may be because he already had a mental model of those levels, having worked with the operating system for so long. EN2, on the other hand, made most of his hypotheses at the domain and program model level, indicating the earlier stage of the enhancement task (where and how to make the enhancement).

EN1 and EN2 also differed significantly in what types of hypotheses they made at the domain-model level. None of the hypotheses types are used by both subjects. This reflects both the different stages of the enhancement task (e.g., EN2 hypothesizes about the location to add functionality (OPH10), something EN1 has already solved) and the amount

Table 7. Total and relative frequencies for hypotheses by model

Observation and analysis	Top–down model	Program and situation models			Total of hypotheses
		Combined	Situation	Program	
EN1 observations	2	34	4	30	36
enhancement	6%	94%	11%	83%	100%
EN2 observations	16	12	1	11	28
enhancement	57%	43%	4%	39%	100%
Total observations	18	46	5	41	64
enhancement	28%	72%	8%	64%	100%
Chi-square results	$\chi^2 = 56{\cdot}9$ $p < 0{\cdot}0001$		$\chi^2 = 20{\cdot}8$ $p < 0{\cdot}0001$		

of prior work with the code (e.g., EN2 makes most of the domain level hypotheses regarding the domain functionality of the software, EN1 has none like that—he already knows).

The few hypotheses at the situation-model level do not allow much interpretation, except that the types are consistent with either debugging or knowledge acquisition. At the program model level, EN1 makes most of his hypotheses about code correctness and cause or location of error (SYSH16). Since he was debugging an enhancement, this is to be expected. The second and third most frequent types of hypotheses concerned variable values (SYSH7) and statement execution order (SYSH6). This is similar to behaviour found during corrective maintenance where the three most frequent hypotheses at the program level were SYSH6, SYSH16 and SYSH7 in that order (von Mayrhauser and Vans, 1997b). Conversely, EN2's program model hypotheses reflect his desire to find out where and how to add the enhancement. Not surprisingly, the most frequent hypotheses made were about the location to add code (SYSH18) and what a given function or procedure does (SYSH2).

Next, we consider the nature of the hypotheses made (*what*, *how* and *why* hypotheses and whether they were confirmed, abandoned or failed). Table 8 gives the total number of hypotheses in each of these categories for each subject and cumulatively for all subjects. A chi-square test also indicates significant differences between the two subjects, both with respect to the nature of the hypotheses and the type of resolution ($p < 0{\cdot}016$). More than twice as many hypotheses are confirmed, than failed or were abandoned. This may be due to both subjects being experts in application domain and language/platform. On the other hand, accumulated knowledge about the software could play a role in the few abandoned hypotheses for EN2 versus EN1. Knowing more about the product may enable a programmer to avoid hypotheses he or she must later abandon.

The majority of the hypotheses were of type *what*. Letovsky (1996) defines *what* hypotheses as those that conjecture about what something is or does. Slightly fewer (26)

Table 8. Distribution and disposition of hypotheses by category

Observations and results	Hypothesis category			Total hypotheses	Hypothesis disposition		
	What	How	Why		Confirm	Abandon	Fail
EN1	21	9	6	36	21	3	12
EN2	10	17	1	28	16	9	3
Total	31	26	7	64	37	12	15
Chi-square results	$\chi^2 = 9{\cdot}07$ $p < 0{\cdot}011$				$\chi^2 = 8{\cdot}2$ $p < 0{\cdot}016$		

how hypotheses (conjectures about the way something is accomplished) and only seven *why* hypotheses (conjectures about the objective of an action or design choice). While the subjects voiced a significant number of *what* and *how* hypotheses, they had few *why* hypotheses. This is similar to our findings for corrective maintenance, but could also be due to their expertise (both programmers on the enhancement tasks and three of the four corrective maintenance programmers were domain experts). It would be interesting to investigate whether more accumulated knowledge about the software is related to more *why* hypotheses (EN1 makes six while EN2 makes only one).

4.4. Information needs

Information needs are developed by analysing each action type for the kind of information it needed or searched for. Table 9 summarizes the results of this analysis in six columns.

The first column in Table 9 provides a code for the information need. The second column describes the information for which the software maintainer is looking. The third column lists the action type codes (associated with a specific model) for which the information is needed. The fourth column lists cumulatively how often the subjects had a need for this information. Columns five and six provide detailed results for EN1 and EN2 individually. These results help to illustrate the type of work and to prioritize solutions that would facilitate it.

EN1 required mostly top–down and program model information. Most of the information is related to debugging. The major activity during the enhancement task was debugging new code, so it is reasonable that EN1 was looking for information indicating causes of errors. Debugging an enhancement requires understanding the use of variables if anomalies are to be found (I4). And, because diagnosis of the bug is an important part of the job, programmers are likely to have to look in various places and remember what they looked at (I9). This describes EN1's information needs. Isolating bug behaviour (I73) and identifying the location of change (I67) are also important when debugging an enhancement.

EN2's information needs were a little different. Classifying related code by application function (I43) is the most important information need, followed by work history (list of

Table 9. Enhancement information needs

Code	Information need	Action codes	Subjects total	EN1	EN2
I9	List of browsed locations	SYS4, SYS10, SYS13, SYS17, SYS21, SIT11, SITKNOW, OP2, OP3, OP7, OPKNOW	11	6	5
I4	Location and uses of identifiers	SYS3, SYS8, OP2, OP20	9	7	2
I67	Location of where to put changes	SYS2, SYS11, SYS12, SYS17, SYS20, SYSKNOW, OP3, OP20	9	5	4
I61	Connected domain–program–situation model knowledge	SYS1, SYS3, SYS10, OP2, OP20	7	4	3
I43	A general classification of routines and functions so that if one is understood the rest in the group will be understood	SYS8, SYS11, SIT4, OP3, OP4, OP14, OP15, OP20	6	0	6
I7	Domain concept descriptions	SYS2, SYS10, SYS11, SIT1, SIT6, OP3	5	1	4
I73	Bug behaviour isolated	SYS3, SYS8, SYS12	5	5	0
I17	Location of desired code segment	SYS3, SYS11, SYS15, OP2, OPKNOW	5	2	3
I68	List of issues/decisions considered during design	SYS4, SYS7, SYS8, SIT4, OP3, OP15	4	0	4
I25	Exact location to set breakpoint	SYS18, SYSKNOW	4	3	1
I2	List of routines that call a specific routine	SYS11, SYS12, OP11	3	0	3
I24	List of executed statements and procedure calls, variable values	SYS8, OP11	3	2	1
I62	Predefined (constant) variables and values	SYS11, SYS15, OP11, OP20	3	0	3
I22	History of past modifications	SYS8, SYSKNOW	2	1	1
I70	State of system when crashed	SYS12, SYS18	2	2	0
I30	Where variable is toggled, when and why, where passed to and why	SYS11, OP20	2	0	2
I65	Assembly language code segment number (machine code)	SYS2, SYSKNOW	2	2	0
I1	Variable definitions, including why necessary and how used, default values and expected values	SYS8	1	0	1
I3	Highlighted begin/ends of control blocks	SYS17	1	1	0
I14	Call graph display	OP6, OP15	1	0	1
I42	Utility function definitions and comments explaining why it was rewritten	SYS7, SIT4, OP17	1	0	1
I32	If common objects are not used in traditional way, e.g., nil or null	SYS8, SYS12	1	0	1
I36	Sequence of locations where ID is used	SYS8	1	1	0
I79	Ripple effect—procedure affected by change, include port effects, dependencies in make file	OP1, OP20, SYS15	1	0	1
I13	Conditions under which a branch is taken or not. Include variable values	SYS2	1	0	1

Table 9. Continued

Code	Information need	Action codes	Subjects total	EN1	EN2
I41	Call graph with extraneous information not relevant elided	SYSKNOW	1	0	1
I48	Code formatted in expected way	SYS21	1	1	0
I21	Organized functions into categories in which functions are related	SYS10	1	1	0
I46	What kind of architecture-dependent code currently exists and is functional, and what architecture hooks exist so code can be easily added	OP20	1	0	1
I77	Dataflow trace	SYS7	1	0	1

browsed locations (I9)). Domain concepts (I7), design issues (I68) and where to put the changes (I67) representing the enhancement are next. EN2's information needs reflect the nature of the early stage of making an enhancement.

Cumulatively, the top three information needs are information on work history (I9—list of browsed locations), where to put code changes (I67), and the location and use of identifiers (I4). The first speaks of possible memory overload during the task; the other two appear specific for enhancement tasks as this pattern does not hold for corrective maintenance *per se* (von Mayrhauser and Vans, 1997a).

5. TOWARD POSSIBLE IMPROVEMENTS

These two programmers, who worked on different aspects of an enhancement task and brought different knowledge about the software to the task, behaved very differently. We saw information needs they had in common and information needs in which they differed. Information needs they had in common are important, because they point out that, regardless of the stage of the enhancement task, regardless of the accumulated knowledge about the software, and despite individual differences, enhancement work needs this information. From a practical point of view, documentation, tools, guidelines and training should try to provide this information in an easily accessible manner. Consider the three most frequent information needs both programmers had in common.

List of browsed locations—I9. Keeping work history at the program level is well within the realm of current tool technology. This is the on-line version of placing coloured paper clips or sticky tape flags on paper pages to remember the work history. We suggest that the using programmer control over the marking of such locations. The issue becomes more involved when we discuss information that spans a variety of documents and levels of abstraction.

Possible tool support could provide bookmarks in chronological order to record work history in a hypertext manner. Such tools must be capable of crossing document boundaries

(code, design, enhancement specification, etc.) and provide a playback capability. This would expedite tracking one's work history better than before and reduce the cognitive load posed in working with large software systems. For this to work well, most documents must be on line.

Documentation should be organized so that less must be remembered about the work history. This reduces the number of browsed locations, alleviating the problem of cognitive overload. Cross-referencing of related information is important. If documents are on line, documents should be organized to show related information together to make immediate cross-referencing possible. It is also helpful to organize documents by key concepts and to follow then from user manual, functional specification, design, etc. to code to give full traceability.

Training should emphasize key concepts and follow them from application-domain level to code. This means grouping and teaching information at the domain, situation and program level by key concept. Enhancement work flows from the specification of a new functionality through to code. Knowing 'what is related' is likely to make the work more focused and require fewer 'fishing expeditions'.

This approach will not be feasible in all cases since it involves a major reorganization of training materials and project documentation. In that case, it is useful to train the programmers in how to keep track of the work history, if necessary, manually. It can be as simple as keeping a scratch pad at one's side to keep track of open questions and what has been done to investigate them, and how they have been resolved.

Guidelines should emphasize that code should be documented so that cross-referencing is possible and requires less browsing. During an enhancement task, the programmer should identify the sections of code, design and functional specifications, as well as domain concepts that are involved, and keep track of them so they can easily be recalled and revisited.

Location where to put changes—I67. In a complex software system, determining where the code needs to be changed can be a daunting task for programmers who are unfamiliar with the system. A significant amount of time must be spent in understanding the existing system structure before new code can be introduced. Thus, identification of specific areas where the code is expected to change would greatly reduce the amount of time spent in understanding the existing system structure.

As before, tools that are capable of cross-referencing related functional, design and code areas would make it easier to identify where code changes ought to be made. High level representations of what is achieved in which portion of the code, identification of key design and code objects and where they can be found in the code, all support finding the answer to this information need.

Documentation should include a high-level road map of the system structure that identifies functional areas and the specific procedures and functions that implement the functionality. Using known software architecture patterns, either domain-specific ones like layers in an operating system, or generic ones, such as discussed by Shaw and Garlan (1996), can speed the understanding of the existing system. A precise description of the enhancement and its relationship to existing functionality is also important.

Successful and efficient enhancement is related to an understanding of the existing

system. The minimum training should include key related domain concepts, structure of the software and the system, use of the system, operating environment, and any tools necessary to complete the enhancement (e.g., debuggers, editors, etc.).

Guidelines should require documenting the system structure both from a domain and module-as-coded perspective. An enhancement request should include detailed references to existing functionality. This would then make it easier to identify possible parts within the software system that either could change or be affected by the change. Having examples of typical system use also helps to identify to which existing features an enhancement request might be related.

Connected domain–program–situation model knowledge—I61. Cross-referencing between levels of information and knowledge was a key information need for both programmers as noted earlier, and provides a further rationale for the suggestions made earlier. More can be done to improve cross-referencing.

Tools should provide (hypertext or web-based) cross-reference links from code to associated algorithm and domain information. Automated documentation assistance might offer templates for describing main program components at various levels of detail from groups of modules down to classes, methods and functions.

Documentation should contain cross-referenced information at the functional, algorithmic and code level. Training for a system should include related application domain information, algorithms and programming approaches used. Knowledge of the high-level structure of the software product should be acquired early to provide a structure into which new knowledge can be integrated easily. We assume that the programmers are familiar with the programming language and the platform—otherwise this, too, must be taught. Guidelines should require structuring information for easy cross-referencing.

6. CONCLUSIONS

Software enhancement is a frequent activity during software evolution. We observed two experienced professional programmers while they were enhancing software and analysed their behaviour. The goal was to answer several questions about how programmers go about enhancing software, their work process and their information needs. Answers can be summarized as follows:

(1) *Actions*. Enhancement work is done predominantly at the program and situation model level, even in the beginning, when decisions are made how to make the enhancement and where to put it. Using accumulated knowledge about the software or acquiring it are key activities. Detailed action types differ depending on the stage of the enhancement task.
(2) *Process*. Like other maintenance tasks, enhancement tasks require frequent switching between levels of abstraction. These switches are multidirectional, consistent with the Integrated Comprehension model (von Mayrhauser and Vans, 1995a). Work duration (in terms of number of actions) at a given level and number of switches between levels differs. We hypothesize that this is due to the amount of accumulated knowledge about the software. The early stage of enhancement also requires

significantly more cross-referencing between domain level, situation level and program level information.

(3) *Hypotheses*. Hypothesis-making behaviour differed between the programmers. EN1 made, almost exclusively, hypotheses about program and situation-model level behaviour, while EN2 made most of his at the domain level. This is likely related to the stage of the enhancement task (planning what to do and how to do it versus debugging the enhancement after it has been made). Differing amounts of prior work with the code should also be a factor. EN2 was a domain expert, but had not worked with the code before. It seems logical that he would approach the problem starting from his own expertise, i.e., domain knowledge, hypothesize about the domain level functionality of the software, and cross-reference into the code from there. The programmers also differed in the types of hypotheses made, reflecting differences in their focus.

Hypothesis resolution happens predominantly through confirmation or failure. EN1 in particular abandons only one hypothesis. It is not clear whether or not this is a function of his extremely strong knowledge of the code. His behaviour supports Brooks' theory on hypotheses (Brooks, 1983), although none of our other observations do (von Mayrhauser and Vans, 1996b, 1997a). Programmers appear to make few 'why' hypotheses and the programmers in the present study were no exception. EN1, however, made many more than EN2. Thus, hypothesizing about 'why' something was done might be related to the accumulated knowledge about a software product. This should be investigated further. We expected more *why* hypotheses during enhancements tasks because enhancements need to fit with the development approach of the existing product. Knowing why software is built in a certain way appears useful, even necessary, for making good enhancement decisions. So it is a bit surprising that so few hypotheses were why hypotheses.

(4) *Information needs*. In the earlier stages of the enhancement task, a programmer needs both domain- and code-related information. During integration of the enhancement into the code and its validation, much more code-related information is sought. Much of this latter information can be provided with current static analysis and dynamic debugger technology. Memory overload appears to be an issue for both subjects during the enhancement task. Both needed information on their immediate work history in a hypertext manner in terms of what they had looked at or browsed before. Since the information was not necessarily only at the code level, this would require tool support that can deal with a variety of code and documents.

(5) *Model support*. The analysis of the observations made in this study adds additional confirmation of the Integrated Comprehension model.

Acknowledgements

We thank the *Journal's* reviewers for their helpful suggestions.

References

Brooks, R. (1983) 'Towards a theory of the comprehension of computer programs', *International Journal of Man–Machine Studies*, **18**(6), 543–554.

Howe, A. E. and Cohen, P. R. (1994) 'Detecting and explaining dependencies in execution traces', in Cheeseman, P. and Oldford, R. W. (Eds), *Selecting Models from Data: Artificial Intelligence and Statistics* IV, Springer-Verlag New York Inc., Secaucus, NJ, pp. 161–182.

Howe, A. E. and Somlo, G. (1997) 'Modeling discrete event sequences as state transition diagrams', in Liu, X., Berthold, M. and Cohen, P. (Eds), *Advances in Intelligent Data Analysis*, Springer-Verlag, Berlin, to appear.

Letovsky, S. (1986) 'Cognitive processes in program comprehension', in Soloway, E. and Iyengar, S. S. (Eds), *Working on Empirical Studies on Programmers*, Ablex Publishing Corporation, Norwood, NJ, pp. 58–79.

Littman, D. C., Pinto, J., Letovsky, S. and Soloway, E. (1986) 'Mental models and software maintenance', in Soloway, E. and Iyengar, S. S. (Eds), *Workshop on Empirical Studies of Programmers*, Ablex Publishing Corporation, Norwood, NJ, pp. 80–98.

Pennington, N. (1987a) 'Stimulus structures and mental representations in expert comprehension of computer programs', *Cognitive Psychology*, **19**(3), 295–341.

Pennington, N. (1987b) 'Comprehension strategies in programming', in Olson, G. M. *et al.* (Eds), *Empirical Studies of Programmers: Second Workshop*, Ablex Publishing Corporation, Norwood, NJ, pp. 100–113.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Persectives on an Emerging Discipline*, Prentice-Hall International, Englewood Cliffs, NJ, 242 pp.

Soloway, E. and Ehrlich, K. (1984) 'Empirical studies of programming knowledge', *Transactions of Software Engineering*, **SE-10**(5), 595–609.

Soloway, E., Adelson, B. and Ehrlich, K. (1988) 'Knowledge and processes in the comprehension of computer programs', in Chi, M. *et al.* (Eds), *The Nature of Expertise*, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, pp. 129–152.

Vessey, I. (1985) 'Expertise in debugging computer programs: a process analysis', *International Journal of Man–Machine Studies*, **23**(5), 459–494.

von Mayrhauser, A. and Vans, A. M. (1993a), 'From program comprehension to tool requirements for an industrial environment', in *Proceedings of the 2nd Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, CA, pp. 78–86.

von Mayrhauser, A. and Vans A. M. (1993b) 'From code understanding needs to reverse engineering tool capabilities', in *Proceedings of the 6th International Workshop on Computer-Aided Software Engineering (CASE93)*, IEEE Computer Society Press, Los Alamitos, CA, pp. 230–239.

von Mayrhauser, A. and Vans, A. M. (1994) 'Comprehension processes during large scale maintenance', in *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 39–48.

von Mayrhauser, A. and Vans, A. M. (1995a) 'Industrial experience with an integrated code comprehension model', *Software Engineering Journal*, **10**(5), 171–182.

von Mayrhauser, A. and Vans, A. M. (1995b) 'Program understanding: models and experiments', in Yovits, M. C. and Zelkowitz, M. V. (Eds), *Advances in Computers, Vol. 40*, Academic Press, Inc., Troy, MO, pp. 1–38.

von Mayrhauser, A. and Vans, A. M. (1996a) 'On the role of program understanding in re-engineering tasks', in *Proceedings of the 1996 IEEE Aerospace Applications Conference*, IEEE Computer Society Press, Los Alamitos, CA, pp. 253–267.

von Mayrhauser, A. and Vans, A. M. (1996b) 'On the role of hypotheses during opportunistic understanding while porting large scale code', in *Proceedings of the 4th Workshop on Program Comprehension*, IEEE Computer Society Press, Los Alamitos, CA, pp. 68–77.

von Mayrhauser, A. and Vans, A. M. (1996c) 'Identification of dynamic comprehension processes during large scale maintenance', *Transactions on Software Engineering*, **22**(6), 424–438.

von Mayrhauser, A. and Vans, A. M. (1997a) 'Program understanding needs during corrective maintenance of large-scale software', in *Proceedings COMPSAC97*, IEEE Computer Society Press, Los Alamitos, CA, to appear.

von Mayrhauser, A. and Vans, A. M. (1997b) 'Hypothesis-driven understanding processes during corrective maintenance of large scale software', in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, to appear.

Authors' biographies:

Anneliese von Mayrhauser is a Professor of Computer Science at Colorado State University and Director of the Colorado Advanced Software Institute. She has been active in the IEEE in organizing professional conferences and in serving as Vice President. She has published on software testing, software metrics, software maintenance, reliability and performance modelling. Dr von Mayrhauser received a Dipl.-Inf. degree in Informatik (1976) from the Technical University in Karlsruhe, and an M.A. (1978) and Ph.D.(1979) in Computer Science from Duke University. E-mail: avm@cs.colostate.edu

A. Marie Vans is a Software Design Engineer at Hewlett-Packard Corporation in Colorado. She is interested in and has published on program comprehension. Dr Vans received a B.S. in Computer Science (1989) from the California State University in Sacramento, California, and an M.S. (1992) and Ph.D. (1996) in Computer Science from Colorado State University. E-mail: mariev@fc.hp.com

Adele E. Howe is an Assistant Professor in the Computer Science Department at Colorado State University. Her research areas include AI planning, agent architectures, information gathering on the world wide web, and evaluation and modelling of knowledge-based computer systems. She received a B.S.E. in Computer Science from the University of Pennsylvania and an M.S. and Ph.D. in Computer Science from the University of Massachusetts. E-mail: howe@cs.colostate.edu